第3章 chapter 3

栈与队列

本章导读

栈和队列是两种特殊的线性结构，是线性表特例。在线性表上的插入、删除等操作一般不受限制，而在栈和队列上的插入、删除操作会受某种限制，对它们来说，插入和删除运算均是对首尾两个元素进行的。

本章主要介绍栈和队列的逻辑特征及其在计算机中的存储表示、栈和队列的基本运算的算法描述，以及栈和队列的应用。

教学目标

本章要求掌握以下内容。

- 栈的基本概念和栈的基本运算。
- 栈的应用。
- 队列的基本概念和队列的基本运算。
- 队列的应用。

3.1 栈

3.1.1 栈的定义

栈(Stack)是限定仅在表的一端进行插入或删除操作的线性表。在表中允许进行插入或删除操作的一端称为栈顶(Top)，而另一端称为栈底(Bottom)。栈的插入和删除操作分别称为进栈和出栈。进栈是将一个数据元素存放在栈顶，出栈是将栈顶中的元素取出。若给定栈 $S=(a_1,a_2,\cdots,a_n)$，如图 3-1 所示，a_1 是栈底元素，a_n 是栈顶元素。由于只允许在栈顶进行插入和删除操作，因此栈的操作是按“后进先出”的原则进行的。栈又称为后进先出表，即 LIFO(Last In First Out)线性表。不含元素的栈称为空栈。

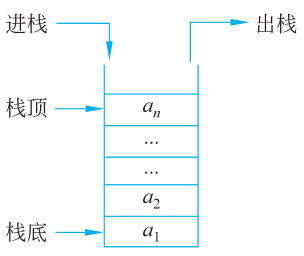


图 3-1 栈结构示意图

日常生活中有许多类似栈的例子，如洗盘子时，把洗净的盘子一个接一个地向上放（相当于进栈），取盘子时，则是从上面一个接一个地向下拿（相当于出栈）。

栈的基本运算有 4 种。

- 初始化栈：将栈置为空栈，不含任何元素，只建立栈顶指针。
- 进栈：向栈顶插入一个新的元素。
- 出栈：删除栈顶元素。
- 判栈空：判断一个栈是否为空栈。

3.1.2 栈的顺序存储及其基本操作的实现

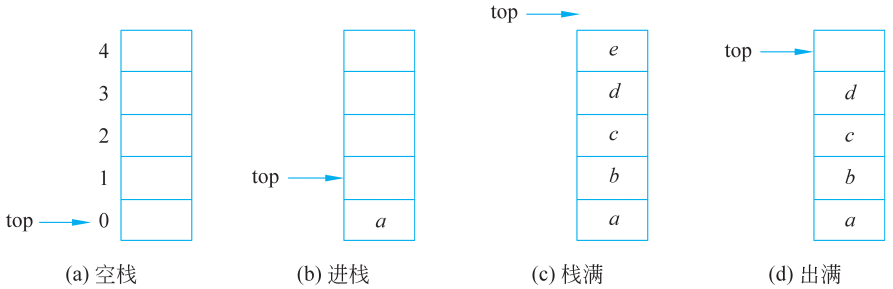
栈的顺序存储结构，简称顺序栈，它是用一组地址连续的存储单元依次存放自栈底到栈顶的数据元素。栈的 Python 语言的类定义描述与顺序表的类定义描述类似。

```
class Stack(object):
  def __init__(self,size):
    self.MAX = size                    #栈的大小
    self.s = []                        #用列表存储栈里的元素
    self.top = 0                       #初始化，若 top=0，则为空栈
```

在这个描述中，列表 s 用于存放栈中的数据元素，栈中可存放数据的最大容量为 MAX，栈顶指针为 top。

在这里，由于栈顶的位置经常变动，所以要设一个栈顶指针 top，用于指向下一次进栈的数据元素的存放位置。当栈中没有数据元素时，栈是空栈，这时 top＝0。当栈中放满元素时，top ＝ MAX，表示栈满。若再有数据元素进栈，栈将溢出，称为“上溢”（overflow），这是一个错误状态；反之，当 top＝0 时，若再进行出栈操作，则会发生“下溢”（underflow）。在应用中，上溢和下溢通常作为控制程序转移的条件。下面以一个例子说明栈的操作。

设有一个栈 $S=(a,b,c,d,e)$，则 MAX 为 5，栈的顺序存储结构如图 3-2 所示。其中，图 3-2(a)是空栈；图 3-2(b)是进栈一个元素 a 之后的栈；图 3-2(c)是在图 3-2(b)基础上连续将元素 b、c、d、e 进栈之后的栈状况，此时是栈满状况，不允许再有元素进栈；图 3-2(d)是在图 3-2(c)基础上出栈一个元素后的栈。由此可见，非空栈中的栈顶指针 top 始终指向栈顶元素的下一个位置。



(a) 空栈　(b) 进栈　(c) 栈满　(d) 出满

图 3-2　栈的顺序存储结构

顺序栈的基本运算有初始化栈、判栈满、进栈、判栈空和出栈，下面分别介绍这几种运算。

1. 初始化栈

初始化栈的算法描述如下。

```
class Stack(object):
  def __init__(self,size):
    self.MAX = size
    self.s = []
    self.top = 0                          #初始化，若 top=0，则为空栈
if __name__ == '__main__':
    s=Stack(50)                           #实例化栈，并把栈的最大容量定义成 50
```

2. 判栈满

在判栈满时，如果栈满，则返回 True；如果栈不满，则返回 False。

```
def stackFull(self):
    if self.top == self.MAX:              #判栈满!!!
        return True
    else:
        return False
```

3. 进栈

进栈是在栈顶插入新的元素。假设栈里原有的数据为 S＝[1,2,3,4,5]，由于进栈时可能遇到栈满（又称“上溢”），导致操作不能进行，所以此时应调用 stackFull()方法判断栈是否满，如果栈满，就返回“真”值，并提示：“栈已经满，不能进行入栈操作!”；如果栈未满，则栈顶指针加 1，插入新元素。进栈的算法描述如下。

例 3-1

```
class Stack(object):
  def __init__(self,size):
    self.MAX = size
    self.s = []
    self.top = 0#初始化，若 top=0，则为空栈
  def stackFull(self):
    if self.top == self.MAX:#判栈满!!!
      return True
    else:
      return False
  def push(self,x):
    if self.stackFull():#进栈之前检查栈是否已满
```

```
            raise Exception("栈已经满,不能进行入栈操作 !")
        else:
            self.s.append(x)
            self.top=self.top+1#push 进去的第一个元素下标为 1
if __name__ == '__main__':
    s=Stack(50)
    s.s=[1,2,3,4,5]
    while True:
        print("----请选择操作方式----")
        print("----1.入栈\n----0.退出")
        number=int(input())
        if number==1:
            x=eval(input("请输入入栈元素"))
            s.push(x)
            print(s.s)
        elif number==0:
            break
```

执行上述程序,结果如下。

```
----请选择操作方式----
----1.入栈
----0.退出
1
请输入入栈元素 45
[1, 2, 3, 4, 5, 45]
----请选择操作方式----
----1.入栈
----0.退出
1
请输入入栈元素 67
[1, 2, 3, 4, 5, 45, 67]
----请选择操作方式----
----1.入栈
----0.退出
```

从栈的入栈操作算法可以看出,该算法的时间复杂度为 $O(1)$。

4. 判栈空

在判栈空时,如果栈空,则返回 True;如果栈不空,则返回 False。
判栈空的算法描述如下。

```
def stackEmpty(self):
```

```
    if self.top == 0:#判栈空
      return True
    else:
      return False
}
```

5. 出栈

出栈是从栈顶删除元素。由于出栈时可能遇到栈空（又称为“下溢”），导致操作不能进行，所以此时应调用 stackEmpty()方法判断栈是否为空，如果栈空，就返回 True，并提示：“栈已经空，不能进行出栈操作!”；如果栈未空，则栈顶指针减 1，返回出栈元素。出栈的算法描述如下。

出栈算法描述如下。

例 3-2

```
class Stack(object):
  def __init__(self,size):
    self.MAX = size
    self.s = []
    self.top = 0#初始化,若 top=0,则为空栈
  def stackEmpty(self):
    if self.top == 0:#判栈空
      return True
    else:
      return False

  def pop(self):
    if self.stackEmpty():
      raise Exception("栈已经空,不能进行出栈操作 ! !")
    else:
      self.top=self.top-1
      return self.s.pop()#利用 Python 的内建函数 pop()实现弹出
if __name__ == '__main__':
    s=Stack(50)
    s.s=[1,2,3,4,5]
    s.top=5
    while True:
        print("----请选择操作方式----")
        print("----1.出栈\n----0.退出")
        number=int(input())
        if number==1:

            print(s.pop())
```

```
        elif number==0:
            break
```

运行上述程序，结果如下。

```
----请选择操作方式----
----1.出栈
----0.退出
 1
5
----请选择操作方式----
----1.出栈
----0.退出
 1
4
----请选择操作方式----
----1.出栈
----0.退出
```

从栈的出栈操作算法可以看出，该算法的时间复杂度为 $O(1)$。

6. 多个栈共享存储空间

对于进栈算法中的“上溢”情况，应设法避免，而避免出现“上溢”的唯一方法是将栈的容量加到足够大。若一个程序使用多个栈，往往事先难以估计每个栈所需容量，在一个栈发生“上溢”时，其他栈可能还留有很多空间，此时可以用移动数据元素位置的方法加以调整，以达到多个栈共享存储空间的目的。

现在以两个栈为例，讨论共享存储空间（列表 s[MAX]，MAX 是预先定义的容量）的方法。可以把两个栈的栈底分别设在给定存储空间的两端，然后各自向中间伸展，如图 3-3 所示，因为此时仅当两个栈的栈顶相遇时才可能发生上溢，这样两个栈之间可以做到互补余缺，使得某个栈实际可利用的最大空间大于 MAX/2。这里设第一个栈自顶向下伸展，第二个栈自底向上伸展，t1 指向第一个栈的栈顶元素的下一个位置，t2 指向第二个栈的栈顶元素的上一个位置，两个栈的初态分别为 t1=0，t2=MAX−1，上溢条件是 t2+1=t1。双栈的 Python 语言描述如下。

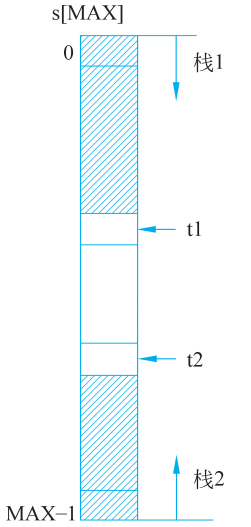


图 3-3 两个栈共享一个存储空间示意图

```
class Doublestack(object):
    def __init__(self, size):
```

```
        self.s = ['null' for i in range(size)]      #s 栈里初始化均填入 null
        self.MAX = size                             #初始化容量
        self.t1 = -1                                #第一个栈的栈顶指针
        self.t2 = size                              #第二个栈的栈顶指针
```

下面介绍双栈的入栈、出栈操作。

1）入栈

向第 i 个栈插入元素 x，若插入成功，则输出入栈之后的结果，否则给出栈满信息，算法如下。

例 3-3

```
#两个栈共享入栈操作
class Doublestack(object):
    def __init__(self, size):
        self.s = ['null' for i in range(size)]      #s 栈里初始化均填入 null
        self.MAX = size                             #初始化容量
        self.t1 = -1                                #第一个栈的栈顶指针
        self.t2 = size                              #第二个栈的栈顶指针

    def StackFull(self):
        if self.t1 + 1 == self.t2:
            print('共享空间栈已满')
            return True
        else:
            return False

    def StackPush(self, flag, data):
        if self.StackFull():
            print('无法再添加数据,栈满')
        else:
            if flag == 1:
                self.t1 += 1
                self.s[self.t1] = data
            else:
                self.t2 -= 1
                self.s[self.t2] = data


if __name__ == '__main__':
    #实例化双栈
    s=Doublestack(10)
    print(s.s)
    while True:
        print("----请选择操作方式----")
```

```
        print("----1.入栈\n----0.退出")
        number=int(input())
        if number==1:
            x=eval(input("请输入入栈元素给 x:"))
            i=eval(input("请输入 1 或者 2 选择向哪个栈进行入栈操作,并将选择赋值
给 i:"))
            s.StackPush(i,x)
            print(s.s)
        elif number==0:
            break
```

运行上述程序,结果如下。

```
['null', 'null', 'null', 'null', 'null', 'null', 'null', 'null', 'null', 'null']
----请选择操作方式----
----1.入栈
----0.退出
1
请输入入栈元素给 x: 12
请输入 1 或者 2 选择向哪个栈进行入栈操作,并将选择赋值给 i: 1
[12, 'null', 'null', 'null', 'null', 'null', 'null', 'null', 'null', 'null']
----请选择操作方式----
----1.入栈
----0.退出
1
请输入入栈元素给 x: 34
请输入 1 或者 2 选择向哪个栈进行入栈操作,并将选择赋值给 i: 2
[12, 'null', 'null', 'null', 'null', 'null', 'null', 'null', 'null', 34]
----请选择操作方式----
----1.入栈
----0.退出
1
请输入入栈元素给 x: 56
请输入 1 或者 2 选择向哪个栈进行入栈操作,并将选择赋值给 i: 1
[12, 56, 'null', 'null', 'null', 'null', 'null', 'null', 'null', 34]
----请选择操作方式----
----1.入栈
----0.退出
```

由前述可知,该算法的时间复杂度为 $O(1)$。

2) 出栈

第 i 个栈出栈算法是,若第 $i(i=1,2)$ 个栈为空,则函数给出栈空的信息,否则显示出栈后栈内数据的变化情况。

例 3-4

```
#两个栈共享出栈操作
class Doublestack(object):
    def __init__(self, size):
        self.s = ['null' for i in range(size)]     #s 栈里初始化均填入 null
        self.MAX = size                            #初始化容量
        self.t1 = -1                               #第一个栈的栈顶指针
        self.t2 = size                             #第二个栈的栈顶指针

    def StackEmpty(self):
        if (self.t1 == -1 and self.t2 == self.size):
            print('共享空间栈为空')
            return True
        else:
            return False

    def StackPop(self, flag):
        if flag == 1:
            self.s[self.t1]='null'
            self.t1-=1
        else:
            self.s[self.t2]='null'
            self.t2+=1

if __name__ == '__main__':
    #实例化双栈
    s=Doublestack(10)
    #出栈前对栈进行初始化操作
    s.s=[34,56,78,'null', 'null', 'null', 'null', 'null',45,90]
    s.t1=2                                         #第一个栈的栈顶指针位置
    s.t2=8                                         #第二个栈的栈顶指针位置
    print(s.s)
    while True:
        print("----请选择操作方式----")
        print("----1.出栈\n----0.退出")
        number=int(input())
        if number==1:
            i=eval(input("请输入 1 或者 2 选择对哪个栈进行出栈操作,并将选择赋值
给 i:"))
            s.StackPop(i)
            print(s.s)
        elif number==0:
            break
```

运行上述程序，结果如下。

```
[34, 56, 78, 'null', 'null', 'null', 'null', 'null', 45, 90]
----请选择操作方式----
----1.出栈
----0.退出
1
请输入 1 或者 2 选择对哪个栈进行出栈操作,并将选择赋值给 i: 1
[34, 56, 'null', 'null', 'null', 'null', 'null', 'null', 45, 90]
----请选择操作方式----
----1.出栈
----0.退出
1
请输入 1 或者 2 选择对哪个栈进行出栈操作,并将选择赋值给 i: 2
[34, 56, 'null', 'null', 'null', 'null', 'null', 'null', 'null', 90]
----请选择操作方式----
----1.出栈
----0.退出
```

由前述可知，该算法的时间复杂度为 $O(1)$。

3.1.3 栈的链式存储及其基本操作的实现

当栈的最大容量事先不能估计时，也可采用链表存储结构，简称为链栈，其结点类型定义为

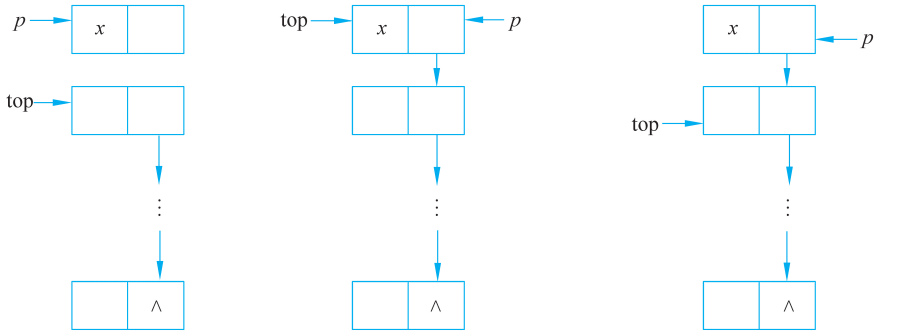
```
class Node(object):
    def __init__(self, data=None):
        self.data = data                #链栈的数据域
        self.next = None                #链栈的指针域
```

设 top 是指向栈顶结点的指针，其初值为空，即在初始状态下，栈中没有任何结点。如图 3-4(a)所示，当把一个数据元素 x 入栈时，先向系统申请一个结点 p，置其数据域值为 x，把 top 结点作为 p 结点的直接后继，top 改指到 p 结点，如图 3-4(b)所示。出栈时，先取出栈顶结点中的数据，然后把该结点链域的值赋给栈顶指针 top，释放该结点，如图 3-4(c)所示。

链栈的主要运算有进栈和出栈。

1. 进栈

当向链栈插入一个新元素时，首先向系统申请一个结点的存储空间，将新元素的值写入新结点的数据域中，然后修改栈顶指针。设要插入的新元素为 x，进栈的算法描述如下。



(a) 链栈*p*结点准备入栈　　(b) 入栈后的结果　　(c) 出栈修改top指针释放*p*结点

图 3-4　栈的插入和删除运算

例 3-5

```
class Node(object):
    def __init__(self, data=None):
        self.data = data                #链栈的数据域
        self.next = None                #链栈的指针域

class Crea_ListStack(object):
    def __init__(self):
        self.top = Node()
        self.count = 0

    #获取长度
    def get_length(self):
        return self.count

    #判断链栈是否为空,链栈没有满栈情况
    def Empty(self):
        return self.count == 0

    #入栈
    def push(self, x):
        node1 = Node(x)
        if self.Empty():
            self.top = node1
        else:
            node1.next = self.top
            self.top = node1
        self.count += 1

    #输出栈里的元素
    def show_stack(self):
```

```
        s = []
        if self.Empty():
            raise IndexError('栈是空的')
        else:
            j = self.count
            p = self.top
            while j > 0 and p:
                s.append(p.data)
                p = p.next
                j -= 1
            print(s)
if __name__ == '__main__':
    s=Crea_ListStack()
    while True:
        print("----请选择操作方式----")
        print("----1.入栈\n----0.退出")
        number=int(input())
        if number==1:
            x=eval(input("请输入入栈元素给 x:"))
            s.push(x)
            s.show_stack()
        elif number==0:
            break
```

上述程序的运行结果如下。

```
----请选择操作方式----
----1.入栈
----0.退出
1
请输入入栈元素给 x: 34
[34]
----请选择操作方式----
----1.入栈
----0.退出
1
请输入入栈元素给 x: 6
[6, 34]
----请选择操作方式----
----1.入栈
----0.退出
1
请输入入栈元素给 x: 5
```

```
[5, 6, 34]
----请选择操作方式----
----1.入栈
----0.退出
1
请输入入栈元素给 x: 4
[4, 5, 6, 34]
----请选择操作方式----
----1.入栈
----0.退出
1
请输入入栈元素给 x: 3
[3, 4, 5, 6, 34]
----请选择操作方式----
----1.入栈
----0.退出
```

由前述可知，该插入算法的时间复杂度为 $O(1)$。

2. 出栈

出栈时，先取出栈顶元素的值，再修改栈顶指针，最后输出原栈顶结点数据，并显示出栈操作后栈内的数据情况。出栈的算法描述如下。

例 3-6

```
#链栈出栈操作
class Node(object):
    def __init__(self, data=None):
        self.data = data                #链栈的数据域
        self.next = None                #链栈的指针域

class Crea_ListStack(object):
    def __init__(self):
        self.top = Node()
        self.count = 0

    #获取长度
    def get_length(self):
        return self.count

    #判断链栈是否为空，链栈没有满栈情况
    def Empty(self):
        return self.count == 0

    #入栈
```

```
    def push(self, x):
        node1 = Node(x)
        if self.Empty():
            self.top = node1
        else:
            node1.next = self.top
            self.top = node1
        self.count += 1
    #出栈
    def pop(self):
        if self.Empty():
            raise IndexError('栈是空的')
        else:
            self.count -= 1
            x=self.top.data
            self.top = self.top.next
            return x

    #输出栈里的元素
    def show_stack(self):
        s = []
        if self.Empty():
            raise IndexError('栈是空的')
        else:
            j = self.count
            p = self.top
            while j > 0 and p:
                s.append(p.data)
                p = p.next
                j -= 1
            print(s)
if __name__ == '__main__':
    s=Crea_ListStack()
    while True:
        print("----请选择操作方式----")
        print("----1.入栈\n----2.出栈\n----0.退出")
        number=int(input())
        if number==1:
            x=eval(input("请输入入栈元素给 x:"))
            s.push(x)
            s.show_stack()
        elif number==2:
            print("将出栈数据赋给 x")
```

```
            x=s.pop()
            print("输出出栈数据")
            print(x)
            print("输出栈里当前数据")
            s.show_stack()
        elif number==0:
            break
```

上述程序的运行结果如下所示。

```
----请选择操作方式----
----1.入栈
----2.出栈
----0.退出
1
请输入入栈元素给 x: 34
[34]
----请选择操作方式----
----1.入栈
----2.出栈
----0.退出
1
请输入入栈元素给 x: 56
[56, 34]
----请选择操作方式----
----1.入栈
----2.出栈
----0.退出
1
请输入入栈元素给 x: 78
[78, 56, 34]
----请选择操作方式----
----1.入栈
----2.出栈
----0.退出
2
将出栈数据赋给 x
输出出栈数据
78
输出栈里当前数据
[56, 34]
----请选择操作方式----
----1.入栈
```

```
----2.出栈
----0.退出
2
将出栈数据赋给 x
输出出栈数据
56
输出栈里当前数据
[34]
----请选择操作方式----
----1.入栈
----2.出栈
----0.退出
```

由前述可知，该算法的时间复杂度为 $O(1)$。

3.1.4 栈的应用举例

栈是计算机软件中应用最广泛的数据结构之一，比如，编译系统中的表达式求值、程序递归的实现、子程序的调用和返回地址的处理等。下面介绍栈的几个应用实例。

1. 算术表达式的计算

表达式求值是编译系统中的一个基本问题，目的是把人们平时书写的算术表达式变成计算机能够理解并能正确求值的表达方法。

算术表达式中包含算术运算符和算术量；各运算符之间存在优先级，运算时必须按优先级顺序进行运算，先运算级别高的，后运算级别低的，而不能简单地从左到右进行运算。因此，进行表达式运算时，必须设置两个栈：一个栈用于存放运算符；另一个栈用于存放操作数。在表达式运算过程中，编译程序从左到右进行扫描，遇到操作数，就把操作数放入操作数栈；遇到运算符时，要比较该运算符与运算符栈的栈顶。如果该运算符的优先级高于栈顶运算符的优先级，就把该运算符进栈，否则退栈。退栈后，在操作数栈中退出两个元素，其中先退出的元素在运算符右，后退出的元素在运算符左，然后用运算符栈中退出的栈顶元素（运算符）进行运算，运算的结果存入操作数栈中。反复进行上述操作，直到扫描结束。此时，运算符栈为空，操作数栈只有一个元素，即最终的运算结果。

例 3-7 用栈求表达式 6－8/4＋3＊5 的值，栈的变化见表 3-1。

表 3-1 用栈求表达式 6－8/4＋3＊5 的值

步 骤	操作数栈	运算符栈	说 明
开始			开始时两个栈为空
1	6		扫描到“6”，进入操作数栈
2	6	－	扫描到“－”，进入运算符栈
3	6 8	－	扫描到“8”，进入操作数栈

续表

步　骤	操作数栈	运算符栈	说　明
4	6 8	－/	扫描到“/”，进入运算符栈
5	6 8 4	－/	扫描到“4”，进入操作数栈
6	6	－	扫描到“＋”，则“/”，“8”，“4”退栈
7	6 2	－	计算 8/4＝2，进入操作数栈
8			扫描到“＋”，则“－”，“6”，“2”退栈
9	4		6－2＝4，进入操作数栈
10	4	＋	扫描到“＋”，进入运算符栈
11	4 3	＋	扫描到“3”，进入操作数栈
12	4 3	＋＊	扫描到“＊”，进入运算符栈
13	4 3 5	＋＊	扫描到“5”，进入操作数栈
14	4	＋	扫描完，“＊”，“3”，“5”退栈
15	4 15	＋	3＊5＝15，进入操作数栈
16			“＋”，“4”，“15”退栈
17	19		4＋15＝19，进入操作数栈
18	19		表达式的值为 19

2. 函数递归的实现

递归是程序设计中常用的方法之一，栈的一个重要应用是可以实现递归函数（或递归过程）。

例 3-8　通常用来说明递归的最简单的例子是阶乘的定义，它可以表示为

$$n!=\begin{cases}1 & n=0,1\\ n(n-1)! & n>1\end{cases}$$

由定义可知，为了定义 n 的阶乘，必须先定义$(n-1)$的阶乘；为了定义$(n-1)$的阶乘，又必须定义$(n-2)$的阶乘……这种用自身的简单情况定义自己的方式，称为“递归定义”。一个递归定义必须一步比一步简单，而且最后是有终结的，绝不能无限循环下去。在 n 的阶乘定义中，当 n 为 0 或 1 时定义为 1，它就不再用递归定义。根据阶乘的定义，编写的 Python 语言程序如下。

例 3-9

```
#用递归函数求 n 阶乘的值
def Fac(i):
    if i==0:
        return 1
```

```
    else:
        return i * Fac(i-1)#因为 n!=n*(n-1)!,所以直接调用自身
if __name__=='__main__':
    n=int(input('请输入阶乘数:'))
    print('%d!值为 %3d' %(n,Fac(n)))
```

运行上述程序,结果如下。

```
请输入阶乘数: 5
5!值为 120
```

由上述程序的运行过程可知,递归求阶乘的算法时间主要花费在递归调用函数的次数,这与 n 的大小有关,因此该算法的时间复杂度为 $O(n)$。

函数直接或间接调用自身称为函数的递归调用,包含递归调用的函数称为递归函数。若函数 Fac(n)中含直接调用函数 Fac(),则称为直接递归调用。若函数 a 中调用函数 b,而函数 b 中又调用了函数 a,则称为间接递归调用。

实现递归调用的关键是建立一个栈,这个栈是由系统提供的运行工作栈。计算机在执行递归算法时,是通过栈来实现的。在一层层递归调用时,系统自动将其返回地址和每一调用层的变量数据一一记下并进栈。返回时,变量数据一一出栈并且被采用。图 3-5 所示展示了递归调用的执行情况,由此可以看到,由于主函数中调用了 Fac(4),Fac 函数共被调用 4 次,即 Fac(4)、Fac(3)、Fac(2)、Fac(1)。其中 Fac(4)是在 main()函数中调用的,其余 3 次是在 Fac()函数中调用的。在某一层递归调用时,并未立即得到结果,而是进一步向深度递归调用,直到最内层函数执行 $n=1$ 时,Fac(n)才有结果。然后在依次返回时,不断得到中间结果,直至回到主程序得到 $n!$的最终结果为止。

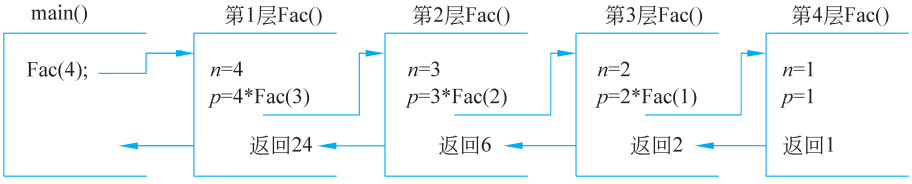


图 3-5 递归调用示意图

实际上,递归是把一个不能或不方便直接求解的“大问题”转换成一个或几个“小问题”,再把这些“小问题”进一步分解成更小的“小问题”,如此分解,直至每个“小问题”都可以直接解决(此时分解到递归出口)。当然,被分解的“小问题”和“大问题”必须具有相同的特征属性。

虽然递归算法简明、精炼,但运行效率较低,时空开销较大,并且某些高级语言没有提供递归调用的语句及功能,因此,在实际应用中往往会使用非递归方法。为了提高程序设计的能力,有必要进行由递归方法到非递归方法的基本训练。通过前面对递归算法的分析,我们已经知道,系统内部是借助栈实现递归的,因此,在设计相应的非递归算法时,也需要人为设置一个栈来实现,具体实现过程将在 5.2.4 节中详细介绍。

3.2 队　　列

3.2.1 队列的定义

队列(Queue)是限定只能在表的一端进行插入，在表的另一端进行删除的线性表。表中允许插入的一端称为队尾(rear)，允许删除的一端称为队首(front)。

假设队列为 $Q=(a_1,a_2,a_3,\cdots,a_n)$，那么 a_1 就是队首元素，a_n 则是队尾元素。队列中的元素是按照 $a_1,a_2,a_3,\cdots,a_n$ 的顺序进入的，出队列也只能按照这个次序依次退出，图 3-6 所示为队列的示意图。队列又称先进先出表(First In First Out，FIFO)。如果队列中没有任何元素，则称为空队列，否则称为非空队列。

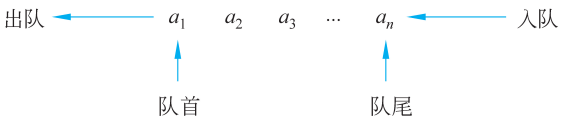


图 3-6　队列示意图

日常生活中有许多队列的例子，如顾客排队购物，排在队伍前面的顾客先买，先离队；排在队伍后面的顾客后买，后离队。队列的基本运算有 5 种。

- 初始化队列：将一个队列设置为空队列。
- 入队列：在队尾插入一个新的元素，也称进队或插入。
- 出队列：在队首删除一个元素，又称退队或删除。
- 取队首元素：得到队首元素的值。
- 判队空：判断队列是否为空队列。

3.2.2 队列的顺序存储及其基本操作的实现

队列的顺序存储结构称为顺序队列(Sequential Queue)。顺序队列与顺序表一样，用一个一维数组存放数据元素。在内存中，用一组连续的存储单元顺序存放队列中的各元素。队列的 Python 语言描述如下。

```
class Queue(object) :
    def __init__(self, size) :
        self. MAX = size                    #定义队列的尺寸
        self.q = []                         #定义队列元素存放的列表
```

在这个描述中，用一列表 q[]表示队列，其中 MAX 表示队列允许的最大容量，用一个指针 front 指示队列的首端，用另一个指针 rear 指示队列的尾端。为方便起见，约定头指针 front 指向队首的前一个位置，尾指针 rear 指向队尾所在位置。

在队列顺序存储结构中，如图 3-7 所示，队首指针和队尾指针是数据元素的下标。在队列为空的初始状态 front=rear=－1。每当向队列中插入一个元素，尾指针 rear 向后

移动一位，即 rear＝rear＋1。当 rear＝MAX－1 时，表示队满。每当从队列中删除一个元素时，队首指针也向后移动一位，即 front＝front＋1。经过多次入队和出队运算，可能出现 front＝rear 的时候，这时队列为空。

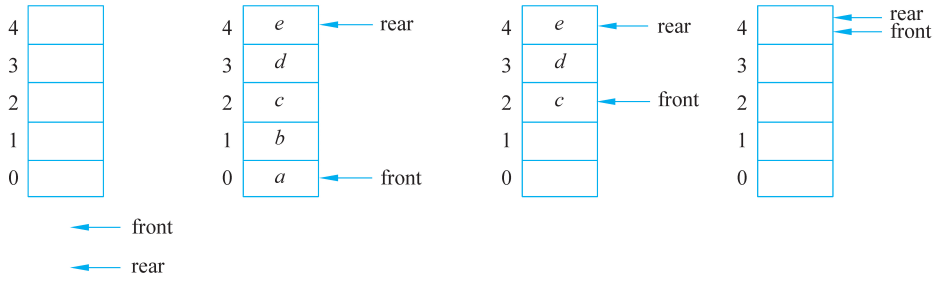


图 3-7 元素出、入队操作示例

下面给出顺序队列中实现入队与出队运算的算法描述。

1. 入队

在顺序队列中入队时，数据元素是从队尾进入的，此时应该先改变队尾指针，使队尾指针指向新元素应插入的位置，即队尾位置，然后将数据插入。由于顺序队列有上界，因此插入时会发生队满的情况，此时应返回队满信息。

例 3-10

```
#队列入队
class Queue(object):
  def __init__(self,size):
    self.MAX = size                    #队列最大尺寸
    self.q = []                        #队列初始化为空列表
    self.front = -1                    #初始化，若 front=-1,则为空队列
    self.rear = -1                     #初始化，若 rear=-1,则为空队列
  def enqueueFull(self):
    if self.rear == self.MAX:#判队列满!!!
      return True
    else:
      return False
  def inqueue(self,x):
    if self.enqueueFull():#入队之前检查队列是否已满
      raise Exception("队列已经满,不能进行入队操作 !")
    else:
      self.rear=self.rear+1            #队尾指针加 1
      self.q.append(x)                 #元素入队

if __name__ == '__main__':
```